The MiniServer Project + Software Engineering

reinhard.budde at iais.fraunhofer.de

Introduction

Many interesting systems are composed of a set of servers using http/https to communicate with each other and their environment ("distributed system"). Each server is a small, closed, robust service. A service exposes features in a REST-like manner. Usually data exchange is based on JSON. Often these services are called "micro services". Usually each service runs a server in a (docker) container of its own to allow separation of the services and scaling if the resource usage changes ("container in a cluster").

This repository provides a simple example to *understand* and *experiment* with this kind of architecture. It tries to avoid surprises and magic. After working with this repository, the lots of magic behind frameworks as Spring Boot, DropWizard etc. should be easier to understand. Then you may decide, to stay with lightweight frameworks as "miniserver" or switch to a more heavyweight frameworks.

There are lots of good tutorials in the web with similar goals. Try them and compare their design with the design here.

This repository located at at https://github.com/rbudde/miniserver.git . The sources in this repository are the basis of industrial-strength software. Thus the examples are not as short as possible, even if they are short :-) A more advanced repository with the same architecture is https://github.com/OpenRoberta/openroberta-lab.git . You may run the software from that repository at https://lab.open-roberta.org .

Tools and why they are required?

- **Java>=8**: the working horse for server programming.

  Of course: other programming languages are ok. They can do the same job. All languages have the same power.

- **Eclipse Photon**: changing the code can be done with any editor. Using an IDE gives you support for completion, formatting, search, call hierarchies, ...

  An alternative is Intellij IDEA, which is also the basis for Android Studio. If you have strong arguments to use another IDE, please contact me.

- **Git**: to maintain the code, you need a version control system. Cvs and svn are out-dated. Though there are other tools as mercurial, Git is my choice without any doubt. Git is powerful (enough) and robust. Git help is very good. Almost all your questions are answered (stackoverflow, e.g.).

  It is nice, that after installation on Windows you get a bash shell for free. Eclipse integration of Git is very good

- **Maven3**: choosing a tool for software configuration management (SCM) is much more problematic. Even after excluding tools as make and ant as too low level(?), there remain a lot of tools, which are discussed controversial. Google for it. All have a lot of pros, but all of them have a lot of cons. You use SCM for the
  - dependency management: declare the artifacts you depend on (frameworks, libraries, ...). Repositories distributed all over the world are accessed to get them. SCM tools can interface with almost all repositories. This is the unproblematic part: all SCM are fine.
  - build process: define how your product is generated. Here SCM tools differ. Some define languages of their own, are extensible with plugins, some are DSLs. As the build requirements for this project are small (generating a library, generate test results), all SCM's will do. So I selected maven as an old, stable tool of industrial strength. Some love it, some hate it.

  Eclipse integration is good, but with a caveat: Maven is a configuration manager based on executing batch jobs. Some features cannot work properly within an IDE, which is based on incremental compilation.

- **Docker**: VMs are well-known and heavy-weight tools for service distribution. Since unix kernels have namespaces and control groups (a long time ago), it is possible to build light-weight containers. Docker is the most popular container management system.

  Easy on linux/unix. If you are win-based, I propse: get a linux box, do almost everything with your win system (programming, testing, ...), commit to a repository, checkout on the linux box and do the Docker-related stuff there. It's easy and you'll get experience with linux boxes, which is fine anyway.

Remarks on working with Java

If project member have to change sources, they *must* agrre on a style, that is enforced automatically, e.g. you may use the Java style definition `rb-compact` from the directory `Misc`. No good project without a formatting style ***all*** members agree upon. I like compact presentation of code, e.g. no separate line for '{', but don't care too much, if *one* style is selected. Furthermore set the save actions in the Java editor panel to:

- Format source code: Format all lines
- Organize imports
- Set as additional actions:
  - Add 'this' qualifier to unqualified field accesses
  - Convert control statement bodies to block
  - Convert 'for' loops to enhanced 'for' loops
  - Remove unused imports
  - Add missing '@Override' annotations
  - Add missing ' @Override' annotations to implementations of interface methods
  - Add missing '@Deprecated' annotations
  - Remove unnecessary casts
  - Remove trailing white spaces on all lines
  - Remove redundant type arguments

Remarks on working with Git

- usage needs experience. That in turn needs time, learning from errors and discussion with co-workers. The tool is great. Read http://nvie.com/posts/a-successful-git-branching-model/ or https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow

- this is the simplest workflow for a group of local developer:
  - **update**: get the work of others. Precondition: git status tells you, that your workspace is "clean"
    git checkout develop
    git pull
  - **branch**: create the feature branch for your work
    git checkout -b myFeature
    ... implement ...
    git add --all
    git commit -m "[ticket-number] expressive text"
  - **rebase**: get the work of others and base your work on this
    git checkout develop
    git pull # skip the remainder if nothing has been pulled
    git checkout myFeature
    git rebase develop
    - fast forward: fine, you're ready to push
    - conflicts: solve the conflicts, practice that, get help!

    a rebase is *only* successful, if no test fails.
  - **push**: make your work available to others. Precondition: successful rebase
    git checkout develop
    git merge myFeature
    git push
  - **be upset**: if between your rebase and your push somebody committed, the rebase must be repeated. If that happens often, this workflow is too simple. Use a more advanced workflow (pull requests, ...).

- clone the repo:
  `git clone https://github.com/rbudde/miniserver.git`
  `git checkout develop`

Remarks on working with Maven

- The **build** is defined in the **`pom.xml`**.To have a clean build, run **`mvn clean install`**

- The **dependencies**: All artifacts are addressed by groupName & artifactName & version" (e.g. "org.slf4j & slf4j-api & 1.7.5"). Dependencies are recursive. You get everything you need in a local cache (".m2"). You can remove the cache or parts of it. The cache is regenerated when needed (internet access!). If in doubt, clear the cache.

- The **dependency management**: a more complex product consists out of a tree of sub-projects originating in a root project. With a dependency management element in the root project you can enforce consistent usage of artifact versions, a must in larger projects. It's easy in this small project, but not a big advantage to have it. Not using it in multi-module projects, is a severe mistake.

- There are **plugins** (too many plugins?) and **declarations** for everything. E.g.:
  - to set the compiler version and the encoding (e.g. the **`maven-compiler-plugin`**)
  - to adapt the build process (e.g. add source directories with the **`build-helper-maven-plugin`**)
  - to integrate tools into the build process (e.g. **`jacoco`**)
  - to retrieve information about the build artifacts, e.g.
    - to tell about newer versions of artifacts: **`mvn versions:display-dependency-updates`**

- whether there are well-known vulnerabilities: **`mvn install -Powasp`**

Software Engineering

Our goals are good, error-free and robust software systems. We try to reach this goals (more or less :-).

The following (low level) principles help to achieve this (we'll not address high level principles as tools and material):

- **encapsulation**: object-oriented software construction, information hiding, single responsibility, separate interface and implementation (class level)
- **DBC**: design by contract, pre- and post-conditions, javadoc for public method based on that, exception handling based on DBC (method level)
- **TDD**: base the design on comprehensive unit and integration tests. Assume that "hard to write tests" means "bad design". Tests describe how to *use* and how *not to use* a class.
- **ARCHITECTURE**: base the system on an understandable architecture, *all* participants have agreed upon. Use tools to check or IDE help to enforce the rules.

Can these principles applied on all level, from classes to containers? Which techniques help to support these principles?

simple classes

- encapsulation, DBC, TDD: the programmer should apply all principles :-)
- Examine examples and their test classes:
  - **`ISecurity`**
  - **`RandomWorker`**
- Notes on TDD:
  - Setting up the test context should be easy. Remember: "hard to write tests" means "bad design" (very often)
  - If the context becomes complex and fragile, try to remove setup costs by using/creating a simple API.
  - What happens, if you don't do that, is visible in the example: HttpClientWrapperTest#testAuth.
    The test method tests HttpClientWrapper#addAuthHeaderOpt, which has complex parameters: credentials and a Http-request.
    See the Javadoc of the method for details
- standards as using java, mvn, git allow processing on almost all machines, e.g. for the CI system bamboo:
  - plan "miniserver" executed after any commit
  - upload quality measures to a sonarqube server
- in Eclipse you can check test coverage (run the eclemma plugin). HttpClientWrapper, for instance, has low coverage. Reason: it contains unused functionality. A very frequent case! Remember: quality of unused functionality degrades.

REST / jersey

- formerly HTTP-based services used heavy-weight objects (http-request, -session, -response). Hard to test.
- today we can achieve encapsulation, DBC and TDD easily by using REST interfaces. In most cases we apply techniques developed in the context of REST, but not the REST-philosophy itself!
- jersey is a REST machine for Java. At startup it scans packages (recursiveley!) and detects service classes (**`@Path`**) with service methods (**`@...`**); later it parses a HTTP-request, selects a method matching an URL pattern (otherwise 404), builds parameters for a call and calls the matching method using reflection.
  - See the server setup in **`de.budde.guice.MiniServerGuiceServletContextListener`** and
  - resource classes in **`de.budde.resources`** as **`HelloWorld#*(...)`** and **`RandomGenerator#getRandom(...)`**
- note, that in **`HelloWorld`** the POST entities are expected as **`String`**. This defect will be remedied in the next section.
- there are many REST clients available as browser plugin or stand alone.

@Provider to avoid hand-written, unsafe boiler-plate code

- **problem 1**: avoid JSON objects except at the system border. REST services get a JSON entity and parse it. Using JSON "deep" in the server impairs typechecking. Getter-setter-beans are possible, but we can do better. Generate the boiler-plate code. Pick up a matching generator. There are some available.
  As example for using JSON for communication between servers the TransportGenerator maven-plugin is used:
  - declaration in the pom

    - Misc/transport.json
    - generated code in de.budde.param, see the pom (source folder: src/main/generated)!
    - contains generated parser/unparser, immutable after construction

- **problem 2**: most REST services get a JSON entity and parse it. This replicates code. Each service needs DBC-based exception handling. This replicates code. Don't repeat yourself.
  Use **`@Provider`**.
    - setup in **`de.budde.guice.MiniServerGuiceServletContextListener`** ...
    - and annotation **`@RandomData`**, provider **`RandomDataProvider`**, ...
    - and the REST service class **`RandomGenerator`** show how to proceed

  use **`@Provider`** for exception handling, too.
    - provider class **`DbcExceptionMapper`** cures exception problems and
    - re-establishs correct DBC-behavior in the context of a distributed system

Summary REST / jersey and provider: encapsulation, DBC and TDD at service level in the same way as with "simple" classes.

See class **`RandomGenerator`** and then class **`RandomGeneratorTest`** as examples, how simple Junit tests for REST services can be.

Dependency injection (DI)

- we want one random number generator in the whole application. A common solution is using the (anti-)pattern *singleton* (see https://en.wikipedia.org/wiki/Double-checked_locking & https://en.wikipedia.org/wiki/Initialization-on-demand_holder_idiom for the unexpected complexity if this is done lazily :-)

```
class RNG {
    private static RNG instance = new RNG();
    public static RNG getInstance() {
        return instance;
    }
    private RNG() {
    }
    ...
    ...
}
```

- singletons are used to avoid parameter passing starting from **`main(...)`** (this would break encapsulation), but add hidden parameter (this breaks DBC and TDD). *A design conflict.*

- use DI (https://en.wikipedia.org/wiki/Dependency_injection e.g.) to avoid singletons

- guice is a good injector, because it is written in Java, and allows injection programmatically. See class **`de.budde.guice.MiniServerGuiceModule`** (Search for comments with **`(1)`** and **`(3)`**)

- DI makes sense if there is (somewhere) an *object life-cycle management*, as in jersey e.g.:
    - HTTP request arrives
    - service class + method are selected
    - service object is instantiated (*)
    - service method of service object is called and creates the response (**)
    - service object becomes garbage

- at (*) and (**) the injector can be used to inject objects.

- the injection is configured in class **`de.budde.guice.MiniServerGuiceServletContextListener`**
- building upon **`de.budde.guice.MiniServerGuiceModule#configure()`** and then

- used in the service classes, e.g. **`de.budde.resources.RandomGenerator`**

Examine the constructor and the methods **`getManyRandoms`** and **`postManyRandoms`** of class **`RandomGenerator`** and the corresponding tests in class **`RandomGeneratorTest`** to see how DI and TDD cooperate: .

summary:
- increases software quality
- after mastering the complexity of the setup everything works like a charm
- programming the injector is flexible and robust (see the security part)
- adding the **`server.properties`** is a nice add-on

Embedded server and integration tests

deploying a server (tomcat, jetty, ...) on a (virtual) machine once and later generating many war's with a web.xml to be deployed on that server is standard and fine. For microservices (micro!) and easy testing embedding the server in an application (with an "own" main(...) is a alternative and done by several "modern" frameworks.

- see the class **`de.budde.jetty.StartServer`**

- the configuration is done programmatically. This is more flexible than using a **`web.xml`**

- the server is started by executing
  `export VERSION='3.0.3-SNAPSHOT'`
- **`java -cp "target/MiniServer-${VERSION}.jar:target/lib/*" 'de.budde.jetty.ServerStarter'`**

- when setting up the injector, static properties (see server.properties) are loaded into the injector. Before that is done, we access runtime properties from the command line and allow overriding the static properties. We use one or more **`-d`** parameter as in
  ```
  java -cp "target/MiniServer-${VERSION}.jar:target/lib/\*"
       'de.budde.jetty.ServerStarter'
       -d self.http=4444 -d person.greeter='craesy, the guinea pig'
  ```
  The **`-d`** parameters add flexibility at runtime!

- TDD: an embedded server allows *integration* tests done with *unit test* frameworks. See class **`de.budde.jetty.RestHttpIT`**:
  - **`@BeforeClass`** and **`@AfterClass`**
  - usage of **`de.budde.util.HttpClientWrapper`**

- setup of many servers is simple. Communication is done using JSON objects:
  - the sender uses the powerful **`HttpClient`** framework (see class **`de.budde.util.HttpClientWrapper`** again)
  - the receiver used standard jersey services
- as an example examine the REST service **`postDelegate`** in class **`RandomGenerator`**. The URL to delegate to is constructed in **`GuiceModule`** (see comment annotated with (2)) and used to configure a **`HttpClientWrapper`** object, that is injected for each call.

- *Side note:* a https connector is created if the property https.port is defined as an integer > 0 in **`server.properties`**. The certificate used is the resource **`/keystore.jks`** . The keystore contains a self-signed certificate and has been generated by calling **`keytool`** (the tool is part of the java SDK):
  `.../keytool -genkey -alias sitename -keyalg RSA -keystore keystore.jks -keysize 2048`

Browser front-ends

- a great combination
  - server based on jersey
  - frontend based on javascript or typescript
  - transport objects are JSON (usually communicated with method POST)

- See, as an example, **`index.html`**, **`lib/adminFn.js`** and **`basics.js`** in **`staticResources`**.
- **`COMM`**, **`WRAP`** and **`LOG`** are modules to achieve robust browser behavior.

- **`bootstrap`** and **`jquery`** are used (potential "upgrades": **`reactjs, angular2`**+**`typescript`**, **`webpack`**)

- run the front-end after starting the server by pointing chrome to **`localhost:1998/static/index.html`**
- **`CTRL-SHIFT-I`**:
  - enhanced reload
  - tooling for inspection and debugging

docker container

- from heavy-weight VM to light-weight container. Docker is the most popular tool.

- based on namespaces (ns) and control goups (cgroups). These are old linux kernel features.

- docker provides:
  - docker command ("frontend")
  - docker demon ("backend")
  - the command communicates with the demon using a socket (**`/var/run/docker.sock`**).

- images and container

- images are build in layers. The incremental build of the layers resembles the way git uses chains of commits.
- Have a look at the file **`Docker/Dockerfile`** in the repository.

- build and run images:
```
export VERSION='3'
docker build -f Docker/Dockerfile -t rbudde/miniserver:${VERSION} .
docker run --name miniserver0 -p 1998:1998 -d rbudde/miniserver:${VERSION}
```

- docker support is available for CI server

- useful commands:

see all images	`docker images`
see the container running	`docker ps`
see all container	`docker ps -a`
stop a container	`docker stop miniserver0`
shell in a running container (debug ...)	`docker exec -it miniserver /bin/bash`
run container, replace entrypoint (debug ...)	`docker run -it --entrypoint=/bin/bash rbudde/miniserver`
rm a container	`docker rm miniserver0`
rm an image	`docker rmi rbudde/miniserver`
rm all terminated container	`docker rm $(docker ps -q -f status=exited)`
rm all images probably obsolete	`docker rmi $(docker images | fgrep '<none>' | awk '{print $3}')`

docker-compose

- for declaration of micro-services (build as container) and their communication: docker-compose, if more control is needed: Amazon AWS with EC2, kubernetes, ... To get a fair overview, google for "orchestration tools".

- docker-compose runs many container as declared in a config file (defaults to "docker-compose.yml").

- All container are started respecting the depends-on relation. But take care: compose *cannot delay* the start of a container until preceding container are *initialized* (simply, because compose doesn't know anything about that). Thus: *always* write a container in a style, that its function does *not depend* on the *availability* of container it depends on. That means: write container *robust* against *failure* of other container. If you did that, the startup is no problem :-).

- For debugging (in non-detatched mode) nice logging is sent to the console. All container are terminated gracefully by ctrl-C.

- For productive use docker-compose is started in detached mode and terminated by a stop command:
```
docker-compose -f Docker/docker-compose.yml up    # start a debugging session.
                                                  # Terminate gracefullly with ctrl-C

docker-compose -f Docker/docker-compose.yml -d up # usual start
docker-compose -f Docker/docker-compose.yml stop  # usual graceful shutdown
```

- run a REST tool and test it.

Self-referential (recursive) Docker

To create the docker image 'rbudde/miniserver:${version}', with a dedicated version, that (later) runs an embedded jetty, you have to install tools as
- git to get the repository
- java to compile the sources
- maven to build the application
- docker to build the image

and of course you have to download the frameworks used. These modifications of the machine running the build process are often unwanted. For instance, if you need installations of tools with different versions, this is not always simple and robust. So the idea is, why not creating an image with the tools and frameworks, and keep the build machine untouched. For different projects you may create different build images and a "framework hell" shouldn't show up.

A further argument, last not least, is, that self-referential systems (Gödel, Turing, ...) are great. The idea is:
- create a docker image 'rbudde/miniserver_gen:1', with the tools,
- run this image as a container and generate the 'rbudde/miniserver:${version}' image.
- then, besides docker, *nothing* has to be installed on the linux box at all.

Running the 'rbudde/miniserver_gen:1' container means:

- use git to checkout the actual sources,
- use maven to create the artifacts,
- use docker to put the artifacts into the 'rbudde/miniserver:${version}' image.

The problem is: we have to store the created image NOT in the running container, because this container will terminate, but in the system that started the container which generated the image.

We achieve this in two steps (and get a highly efficient generation):

- we create the **gen** image. This image contains debian linux, java8, docker, git and mvn installations. It runs "apt-get update & upgrade" to get actual versions.
  Furthermore it clones the miniserver repository and runs mvn, but only to cache all maven plugins and artifacts needed. This is a significant speed-up of further builds. The miniserver repository is removed afterwards. Then we define the shell script **miniServerGen.sh** as entry-point (see the dockerfile for details)
  docker build -t rbudde/miniserver_gen:1 -f Docker/DockerfileGen .

- we run the **gen** image. If the container is started, the entry-point script does what it has to do: checkout the sources, build the system, put it into a container (see the **miniServerGen.sh** script for details).
  The **-v** parameter is the self-referential trick: it tells docker in the running container to use the docker demon of the system starting the run. It runs fast (only the build of the meta image is expensive):
  docker run -v /var/run/docker.sock:/var/run/docker.sock rbudde/miniserver_gen:1 ${VERSION}

- after the run a new image **rbudde/miniserver:${VERSION}** with the matching version number appears in the local registry of the system starting the run magically :-).

This is a great blog discussing docker in docker. You should read it:
http://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/

Possible simple extensions (to be discussed)

- logging with slf4j and logback
- double click avoidance in the frontend
- ticket system for logging and dbc exceptions
- hibernate + hsqldb for logging or logstash or cassandra or kafka + kibana/grafana(?)
- msg enum + i18n